A Case for SmartNIC-accelerated Private Communication

Duckwoo Kim, SeungEon Lee, and KyoungSoo Park
School of Electrical Engineering, KAIST

ABSTRACT

Transport Layer Security (TLS) has become a key building block for private network communication in modern Internet. While recent advancement of CPU has substantially improved the data encryption performance, TLS key exchange still remains the bottleneck for short-lived transactions. Dedicated hardware crypto accelerators promise good performance, but they often require invasive modification of the application due to its inherent architecture of asynchronous processing.

In this paper, we explore a potential for offloading TLS handshake to network interface cards (NICs) with a hardware crypto accelerator. We envision a split TLS processing architecture for TCP that handles TCP connection setup and TLS handshake on NIC while carrying out the remaining operations in the CPU-based host stack. We present our rationale for the design and discuss a set of challenges towards our goal. Our proof-of-concept implementation on existing SmartNIC shows a promising result as it brings 5.9x throughput improvement than that of a single CPU core.

CCS CONCEPTS

• **Hardware** → **Networking hardware**; • **Networks** → **Session protocols**;

KEYWORDS

TLS; Programmable NIC; NIC offloading

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APNet '20, August 3–4, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8876-4/20/08…$15.00
https://doi.org/10.1145/3411029.3411034

ACM Reference Format:

Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. 2020. A Case for SmartNIC-accelerated Private Communication. In *4th Asia-Pacific Workshop on Networking (APNet '20), August 3–4, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3411029.3411034

1 INTRODUCTION

TLS is increasingly popular for private communication in the modern Internet; over 70% of the Internet traffic is encrypted with TLS in 2018 [8] while 96 out of top 100 websites present HTTPS as the default protocol [10]. TLS is not only used by default in HTTP/2 [14], but it has been also employed by various transport or application layer protocols such as QUIC [21], CoAP [13], Skype [30], online meeting services [33, 9], and so on.

The TLS operations are largely divided into key exchange and encrypted data transfer whose relative performance overhead depends on the size of the delivered data [4]. For short messages like Web browsing, the bottleneck typically lies in public key cryptography for key exchange. In contrast, symmetric key cryptography and content hashing for integrity are the main bottleneck for large messages such as online video transfer. Fortunately, the performance for the latter has greatly improved due to recent introduction of AES-NI [11] as even a single CPU core achieves over 20 Gbps for AES on Galois/Counter Mode (AES-GCM) [6] now. However, the performance of public key cryptography has not improved as much as the same CPU core handles slightly over 1500 operations per second with 2048-bit RSA, which degrades the HTTPS performance of short connections by up to 33x from that of plaintext transfer.

In this work, we explore a design space that offloads TLS handshake on TCP to SmartNIC while performing the rest of the operations in the host CPU. We look into if it is beneficial to place a hardware public key accelerator on NIC for scalable TLS key exchange while leveraging AES-NI on CPU for high-throughput data encryption and decryption. This split architecture, which we call SmartTLS, brings a number of benefits. First, it would substantially alleviate the burden of TCP connection setup and TLS key exchange from CPU. The saved overhead includes per-flow state tracking

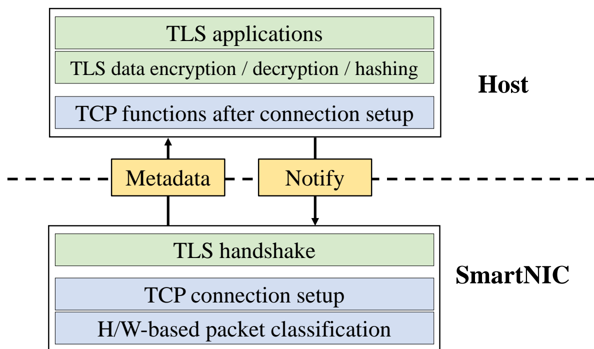

Figure 1: Division of TCP/TLS functionalities between SmartNIC and host stacks

AES mode	1 core	2 cores	4 cores	8 cores
CBC (w/o NI)	2.02	4.05	7.37	13.62
CBC (w/ NI)	4.36	8.50	15.55	28.90
GCM (w/o NI)	1.28	2.52	4.61	8.50
GCM (w/ NI)	20.74	40.90	75.10	138.98

Table 1: AES performance on CPU with 256-bit key. The numbers are in Gbps.

Key size	1 core	2 cores	4 cores	8 cores	PKA
1024-bit	7900	15273	28411	52581	30986
2048-bit	1591	3115	5722	10554	5239
4096-bit	152	296	537	993	743

Table 2: Performance of RSA private key operations on CPU. The unit is # of operations/sec. PKA refers to the performance measured on Mellanox Bluefield NIC.

and DMA operations of small packets between NIC and host memory, which could be significant in case of many concurrent connections. Second, performing TLS key exchange on NIC requires little or no modification of the application unlike an alternative design that employs dedicated hardware like GPUs [19] or ASIC-based crypto accelerators [16, 28]. Dedicated accelerators typically require asynchronous or batch processing of multiple crypto operations for high throughput, which inevitably demands updating the application architecture [32]. In contrast, SmartTLS is transparent to the application, so it can still use synchronous crypto library APIs. Figure 1 summarizes the workload divided between host CPU and SmartNIC in SmartTLS.

Despite the benefits, SmartTLS introduces a set of new challenges. First, NIC must ensure reliable transfer of the packets exchanged during TCP connection setup and TLS key exchange. Implementing reliable data delivery is complex and error-prone as it requires inferring packet loss, implementing timers, and retransmitting packets. This poses a heavy overhead as typical SmartNICs do not have so much computing capacity as CPU. Our approach to tackling this challenge is to make the functionality on NIC stateless as much as possible. We implement stateless TCP connection setup with SYN cookie as in AccelTCP [25], and exploit the client side to initiate any packet retransmission. This greatly simplifies the implementation on NIC and ensures good performance for average case. Second, after TLS handshake, NIC should efficiently forward the incoming packets to the host CPU side. However, even checking the four tuples of a packet on NIC to tell if TLS handshake is done is costly, which potentially degrades the performance. To avoid the overhead, we leverage hardware packet classification on NIC that ensures line-rate packet forwarding performance. While the latency for setting an individual rule in the hardware switch on NIC looks a bit high for now, we believe it will be resolved in the future. Third, it is possible that NIC itself can be overloaded with too many TLS handshakes, which would degrade the performance of the overall system. To address this problem, we enforce opportunistic offloading, which disables the TLS offloading for new connections if NIC is overloaded.

We implement the prototype of SmartTLS with the Mellanox Bluefield SmartNIC [24]. Our preliminary evaluation demonstrates clear benefit with TLS handshake offloading – SmartTLS achieves 13K TLS connections per second with 2048bit-RSA on an octa-core CPU server, a factor of 1.7 improvement over the CPU-only approach. We expect a much higher performance if the NIC employs a state-of-the-art crypto accelerator. We believe that SmartTLS presents a practical design that handles modern TLS traffic in a cost-effective manner.

2 A CASE FOR TLS HANDSHAKE ON NIC

In this section, we present the rationale for offloading TLS handshakes to NIC and discuss the challenges in the system design.

2.1 Why Offloading TLS Handshake?

ECDHE-RSA-AES-GCM-SHA is currently the most dominant TLS cipersuite used by Alexa Top 1 million sites [1]. ECDHE-RSA refers to ephemeral key exchange based on the Elliptic-curve Diffie–Hellman algorithm signed by an RSA private key. It ensures perfect forward secrecy such that the TLS traffic cannot be decrypted even in the future as it uses a one-time key that is deserted after each TLS transaction. Since elliptic-curve algorithms employ much smaller keys

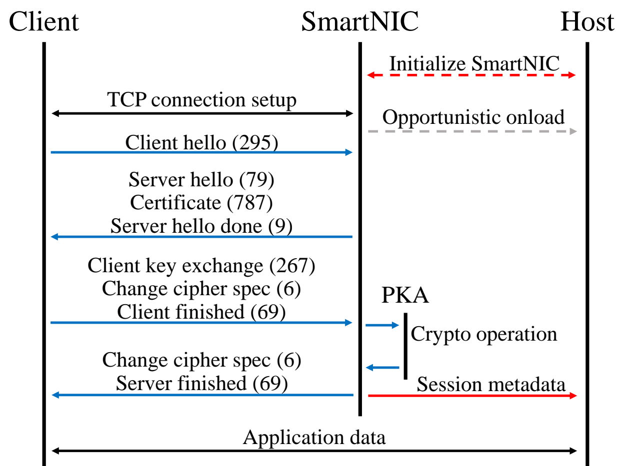

Figure 2: Data flow of an TLS (v1.2) connection on SmartTLS for key exchange with 2048-bit RSA . Values in the parentheses indicate the number of bytes required for each record. SmartTLS coalesces all TLS records in each arrow into a single packet during the handshake.

than RSA's for the same security level, RSA signing remains the most heavy operation in this key exchange algorithm. AES-GCM-SHA means that it uses AES-GCM for authenticated data encryption/decryption and SHA for checking the integrity of key exchange process, respectively. Unlike AES under the cipher blocking chaining (CBC) mode, AES-GCM does not require SHA-based HMAC for packet data integrity, which significantly reduces the overhead in creating a TLS record of application data.

Table 1 shows the performance of AES on recent CPU (Intel Xeon E5-2640 v3 at 2.6GHz) measured with OpenSSL 1.1.1f [1]. We use 256-bit keys and compare the performance of CBC and GCM modes with and without AES-NI. From the table, we find that (1) AES-NI significantly improves the performance for both modes – by up to 2.2x and 16.2x for CBC and GCM, respectively, and that (2) the GCM mode outperforms the CBC mode by 10.2x when AES-NI is enabled. In general, GCM can be implemented more efficiently as it can operate in parallel while it effectively takes advantage of an instruction pipeline. In contrast, encryption in the CBC mode incurs pipeline stalls as each block encryption is completely serialized. We note that even a single CPU core achieves over 20 Gbps with AES-GCM, and an octa-core CPU is enough to reach 100 Gbps for data encryption.

[1]This is the latest version as of writing.

Table 2 shows the performance of RSA private key operations on the same CPU as well as on Mellanox Bluefield SmartNIC [24] equipped with a hardware public key accelerator (PKA). The measurements indicate that one CPU core can handle slightly over 1500 operations per second with 2048-bit keys, but the performance drops by an order of magnitude if we double the key size [2]. Note that NIST currently recommends the RSA key size of 2048+ bits for security but it recommends 3072+ bits if security is required beyond 2030 [2]. Also, the number of sites that adopt 4096-bit RSA is fast growing [1], which implies that the CPU performance for TLS key exchange would be insufficient in the near future. In contrast, PKA shows 3.3x and 4.8x better performance than a single CPU core for 2048-bit and 4096-bit RSA, respectively. This shows that the NIC can save 3.3 or 4.8 CPU cores for handling TLS handshakes.

In summary, modern CPU has relatively higher capacity for encrypting TLS data packets than for performing TLS handshakes. If we can offload TLS handshakes to SmartNIC, we can spend more CPU cycles on AES and SHA operations for higher TLS performance. Fortunately, some SmartNICs [3, 24] already have a hardware PKA module, so we can exploit it for TLS handshake offloading. However, this by no means implies that existing SmartNIC has enough capacity for TLS handshake. Instead, we simply argue that SmartNIC is a great place for a hardware PKA for in-line acceleration of TLS traffic. Given that a recent hardware accelerator achieves 120K operations per second for 2048-bit RSA [28], we believe it is possible to further improve the performance of public key crypto on SmartNIC.

From now on, we discuss the challenges towards our goal and explain how we address them.

2.2 Managing Handshake Complexity

Challenge. Performing TLS handshake on NIC would require complex state tracking during TCP connection setup and TLS key exchange, which poses severe overhead for resource-constrained SmartNICs. Ensuring reliable packet delivery demands inferring packet loss, detecting duplicate ACKs, implementing timers, etc., which would consume considerable compute cycles. This overhead could be large enough to nullify the benefit of hardware-accelerated public key cryptography on SmartNIC.

Approach. Keeping minimal per-flow state is the key to addressing the complexity during TLS handshake. To achieve this goal, SmartTLS breaks down the process of TLS handshake into two phases. First, it ensures *stateless* TCP connection setup with SYN cookie as seen in a recent work [25].

[2]The time complexity of an n-bit RSA private key operation is roughly $O(n^3)$ with square-and-multiply for modular exponentiation [19].

This essentially removes the need for timer-based packet retransmission as well as any state keeping for half-opened connections. Second, SmartTLS exploits the TLS clients for inferring packet loss and retransmission in the key exchange phase. As Figure 2 shows, all the server needs to do is to passively respond to client-side packets during TLS handshake. SmartTLS must keep track of the last sent sequence number as well as the last received ACK, but the server does not have to infer the loss of its own packets as the client would retransmit its previous packet if it does not receive the packet from the server. If necessary, the server retransmits the packets in response to client-side packets.

This design choice optimizes for the average case where packet loss is rare in the real world. This eliminates per-flow timers and packet loss tracking, which enables keeping minimal state - four tuples of a connection with last sequence/ACK numbers, cached TLS packets for integrity calculation and retransmission, along with security parameters (pseudo-random numbers and pre-master/master secret). To guard against the exceptional case where the client becomes unavailable during key exchange, we implement a coarse-grained global timer on NIC that periodically removes the stale connections stuck in this phase for a few seconds.

2.3 Bypassing Embedded System on NIC

Challenge. To benefit from a built-in accelerator on SmartNIC, packets must be forwarded into the embedded system on the NIC. While the embedded system on SmartNIC like Mellanox Bluefield employs a 16-core ARMv8 processor with 16 GBs of memory, its packet processing performance is fairly limited compared to that of a commodity CPU-based system even with a scalable packet I/O library like Intel DPDK [7] [3]. Forwarding all packets through the embedded system would not only lead to latency blowup but it would also degrade the overall throughput.

Approach. It would be ideal to forward only those packets needed for TLS handshake to the embedded system while having all other packets bypass the SmartNIC system. Towards this goal, we explore the idea of harnessing the hardware packet switch on SmartNIC —the Bluefield NIC supports ASAP^2 OvS offload [22] which enables the NIC embedded system to add a traffic control (TC) rule to forward the packets directly to the host system. We use the TC rules for two cases. First, we install a rule that forwards only relevant TLS packets (e.g, with a matching destination port #) to the embedded system at initialization. Then, all other packets would be forwarded directly to the host. Second, whenever TLS handshake for a flow is completed, SmartTLS inserts a new TC rule with its four tuple so that all future packets in the flow bypass the embedded system. When the connection terminates, one can remove the TC rule from the hardware switch. We observe that installed TC rules promise line-rate packet forwarding to the host system.

Unfortunately, the current system (i.e., the Bluefield NIC) fails to achieve fast rule setup/removal nor high throughput with many installed rules. We find that it takes over one millisecond to install or remove a TC rule on the Bluefiled NIC. Also, while it supports up to 40K TC rules, the packet forwarding speed gradually decreases as number of installed rules grows: it drops to half the line rate at 40K TC rules. In the long run, we hope that the hardware switch performance improves, but for the interim, we can install TC rules only for long-running TLS connections.

2.4 Preventing SmartNIC Overload

Challenge. Offloading TLS handshake reduces the computation burden from host CPU, but too many concurrent TLS handshakes would overload the NIC while host CPU is idle. It is desirable to keep the TLS load balanced between host CPU and SmartNIC, but that is difficult to achieve if most of the connections are short-lived.

Approach. To use both computing resources in a balanced manner, we consider opportunistic offloading of TLS handshake. The idea is pretty simple. When the load to the NIC exceeds a "high" threshold, the NIC performs only TCP connection setup while it asks host CPU to continnue with TLS handshake. For this, SmartNIC keeps a simple counter that reflects the load to its crypto accelerator (e.g., it increments when a crypto operation is dispatched and decrements when an output is produced). TCP connection setup is still carried out on NIC as its stateless nature ensures a high throughput even at high load. When the load goes below a "low" threshold, SmartTLS re-enables TLS handshake offloading.

Our approach is similar to [19] on a high level, but it differs in that the offloading unit is entire TLS handshake rather than individual operations. This design choice would minimize state sharing between host and NIC TLS stacks, which simplifies the implementation. It requires no modification on the application as it is a completely transparent functionality.

3 IMPLEMENTATION

Figure 3 shows the overall architecture of the SmartTLS NIC stack that reflects the design choices in Section 2. When a new packet arrives, the hardware packet classifier on NIC determines if the packet should be forwarded directly to the host with its installed TC rules. If it fails to match any rule, the packet is forwarded to the embedded system on NIC. Then, the system looks up the connection state of the

[3]We see 20 Gbps of DPDK packet forwarding performance for 64B packets all 16 ARM cores on the Mellanox Bluefield NIC.

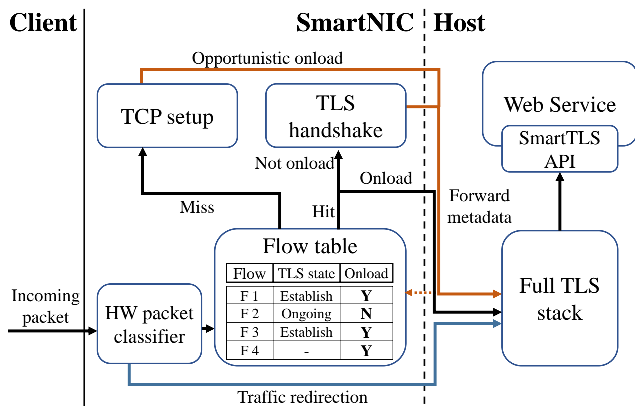

Figure 3: The architecture of the SmartTLS NIC stack

packet in its flow table. If it is a SYN packet whose flow is not found, the system enters into TCP connection setup with SYN cookie. If a matching flow entry is found, the system checks the "onload" flag in the entry, which indicates whether the packets of the flow should be forwarded to the host. If the flag is off (e.g., F2 in the figure), the system carries out TLS handshake with the packet. A flow is typically onloaded when TLS handshake is over but the flag can be set right after TCP connection setup if the NIC is being overloaded (e.g., F4 in the figure).

The NIC passes the TLS metadata along at onloading a flow to the host. The TLS metadata includes four tuples of the flow and TLS parameters for the session (e.g., TLS version and session keys). Then, the host creates a TCP/TLS flow state with a TLS session according to the metadata, and processes all subsequent packets that belong to the same flow. The host should be able to handle TLS handshakes by itself in case the NIC decides to offload its load to the host side. The host TLS stack manages these two separate paths internally by exposing a consistent API to the application layer; the application does not need to care about where the TLS handshake is conducted.

We implement our prototype with the Mellanox Bluefield NIC. The NIC stack includes 6,942 lines of C code (LoC) for packet I/O with DPDK, TCP/TLS flow management, and crypto acceleration with the PKA module. We build the host stack on top of the mTCP user-level TCP stack [18] and write 4,486 LoCs to conform to TLS 1.2 operations. For TLS ciphersuites, we currently implement TLS_RSA_AES_256_GCM_SHA384 and TLS_RSA_AES_256_CBC_SHA, but we plan to support ECDHE for key exchange in the near future.

4 PRELIMINARY EVALUATION

We evaluate the effectiveness of offloading TLS handshake with our SmartTLS prototype. We implement a simple HTTPS web server on top of SmartTLS, and compare the TLS performance with nginx [27] on the Linux TCP stack (kernel version 3.10) as well as on the mTCP user-level stack with OpenSSL 1.0.2 [4]. Note that SmartTLS provides a synchronous crypto API similar to that of OpenSSL, so it should be relatively straightforward to migrate existing applications to using SmartTLS.

Experiment Setup. We measure the performance of one server with four clients. All machines have one Intel Xeon CPU E5-2640 v3 @ 2.60GHz (8 cores) with 32GB of DRAM. The server is configured with a dual-port 25G Mellanox Blue-Field NIC with ARMv8 A72 (16 cores) and 16GB DRAM, but only one of the ports is used for evaluation. All clients are configured with an Intel XL710 40GbE NIC. We configure the SmartNIC on the server to forward all packets directly to the host side for the experiments on Linux and mTCP stacks. For SmartTLS experiments, we have it opportunistically offload TLS handshakes from host CPU.

TLS Handshake Performance. Figure 4 compares the throughputs of TLS connection setup over varying numbers of CPU cores. This experiment shows how many TLS connections can be created for a unit time with TLS key exchange with 2048-bit RSA operations. With a single CPU core, the Linux SSL stack handles slightly over 1000 connections per second, and we observe that the throughput linearly increases to the number of CPU cores. Switching to a user-level TCP stack shows about 8 to 26% performance improvement benefiting from higher efficiency of handling small packets. SmartTLS achieves 6.3K to 13.3K TLS connection setups per second, resulting in a factor of 1.7 to 5.9 improvement over the Linux stack. This confirms that the hardware PKA module on NIC effectively offloads the heavy RSA operations while opportunistic offloading balances the load across the NIC and the host CPU.

Figure 5 compares the 99th % tail latencies of downloading 1-byte file on HTTPS with a single CPU core. When the number of concurrent connections is small, the CPU-only approach shows a better latency due to the packet processing delay on the embedded system on SmartNIC. However, when the concurrency goes beyond 2 flows, the tail latency of the Linux stack quickly blows up more than double the latency of SmartTLS at 16 flows. This indicates that host CPU can easily become the bottleneck for TLS handshake, which would increase the processing latency even for a small number of concurrent flows.

[4]We use a slightly old version due to a compatibility issue with mTCP, but its crypto performance is similar to that of the latest version.

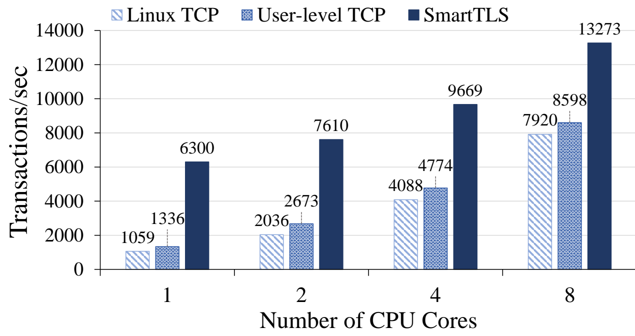

Figure 4: Comparison of TLS handshake throughputs over increasing numbers of CPU cores

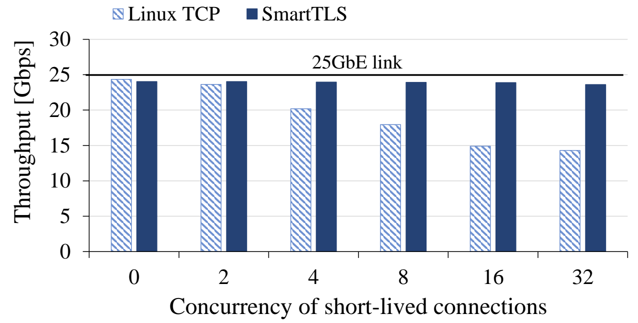

Figure 6: Throughputs for downloading large files in HTTPS with a different number of short-lived connections

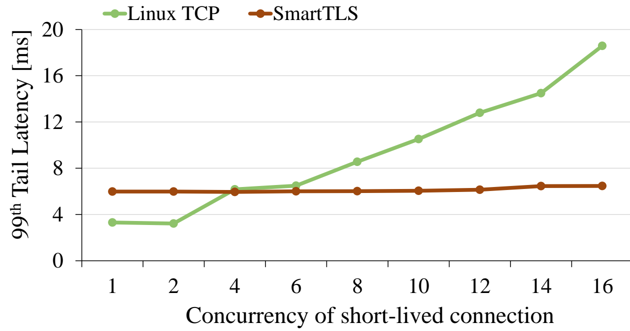

Figure 5: 99th% tail latency of short-lived TLS connections with varying concurrency on a single CPU core

Mixed Workload Performance. We investigate if short-lived connections could interfere with the performance of large-file transfers on TLS. We set up a few flows that transfer large files (4 GB) on HTTPS, which fills up the NIC bandwidth. Then, we start adding a few short-lived connections that download 1-byte file, repeatedly. We use 2048-bit RSA for key exchange and AES-GCM 256bit keys with SHA384 for data encryption and integrity. For this experiment, we use 4 CPU cores that are enough to saturate the NIC bandwidth of 25 Gbps with the given TLS ciphersuite.

Figure 6 shows that the throughput of the large-file traffic degrades as the number of short-lived flows increases. We observe as much as 40% of throughput loss with only 16 concurrent flows. This is due to the heavy contention on the host CPU for AES-GCM and RSA operations. In contrast, when we use SmartTLS, we find little or no performance degradation on the large-file traffic as the TLS handshakes of the short-lived flows are effectively offloaded to the crypto accelerator on NIC.

5 RELATED WORKS

PCIe-based Crypto Accelerators. Hardware crypto accelerators have long been used to supplement CPU for handling TLS traffic. These can be based on general-purpose processors like GPU or FGPA [19, 17, 20, 31] or dedicated devices like Intel QuickAssist Technology (QAT) [16] or Marvel Nitrox security processors [28]. In fact, the performance of recent accelerators is impressive. For example, QTLS reports over 90K TLS handshakes per second with 2048-bit RSA with an Intel QAT adapter [12] while Nitrox V achieves even better performance (120K/sec for 2048-bit RSA) [29]. Obviously, we do not claim that our SmartTLS prototype outperforms these accelerators. In contrast, our main argument is that it is simpler and potentially more cost-effective to place a high-performance PKA module onto NIC rather than employing dedicated accelerators for TLS. This is because using PCIe-based crypto accelerators is often complex as they require asynchronous/batch processing of crypto operations for high performance. Also, the performance benefit is mostly limited to public key operations as the AES performance of modern CPU with AES-NI is either comparable to or even better than that of these accelerators [29, 15].

Offloading Data Encryption and Integrity Computation. Some NICs [5, 23, 26] support offloading AES and SHA operations from CPU to their hardware accelerators while performing TLS handshake with host CPU. This approach is the exactly opposite to that of SmartTLS, but it may be useful when the workload mainly deals with large messages. However, we believe that the effectiveness of offloading data encryption has relatively diminished with the introduction of AES-NI to modern CPU. Since AES-GCM subsumes authenticated data integrity and the CPU overhead for it is fairly low these days, offloading TLS key exchange would be more cost-effective especially when the workload is a mix of large and small transfers.

6 CONCLUSION

We have shown that NIC-offloading of TLS handshake is a viable option that could potentially reduce a large amount of CPU cycles in handling modern TLS traffic. We have built a prototype system with SmartNIC and confirmed the performance benefit with HTTPS traffic. In the mean time, we have also identified a few areas for improvement such as the performance of a hardware switch and a crypto accelerator on NIC. We believe TLS handshake offloading would be a great candidate to be implemented as part of existing NIC offloading schemes such as TSO, LRO, and checksum offloading.

ACKNOWLEDGMENTS

We thank anonymous reviewers of APNet'20 for their insightful comments. This work is supported by the ICT Research and Development Program of MSIP/IITP, Korea, under [Development of an ultra low-latency user-level transfer protocol].

REFERENCES

[1] Alexa Top 1 Million Analysis. https://scotthelme.co.uk/alexa-top-1-million-analysis-february-2019/. Accessed: 2020-05-08.

[2] Elaine Barker. 2019. Recommendation for Key Management: Part 1 – General. Technical report. NIST.

[3] Broadcom Stringray™ SmartNIC. https://www.broadcom.com/products/ethernet-connectivity/smartnic. Accessed: 2020-05-08.

[4] P. Druschel C. Coarfa and D. S. Wallach. 2002. Performance Analysis of TLS Web Servers. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*.

[5] Chelsio T6 ASIC Architecture. https://www.chelsio.com/terminator-6-asic/. Accessed: 2020-05-08.

[6] J. Viega D. McGrew. 2004. The Galois/Counter Mode of Operation (GCM). In *NIST Modes of Operation Process*.

[7] Data Plane Development Kit. https://www.dpdk.org. Accessed: 2020-05-08.

[8] Fortinet Threat Report. https://www.fortinet.com/content/dam/fortinet/assets/threat-reports/threat-report-q3-2018.pdf. Accessed: 2020-05-08.

[9] Google Meet. https://meet.google.com/. Accessed: 2020-05-08.

[10] Google Transparency Report. https://transparencyreport.google.com/https/overview. Accessed: 2020-05-08.

[11] S. Gueron. 2010. Intel Advanced Encryption Standard (AES) New Instructions Set. Technical report. Intel Corporation.

[12] Xiaokang Hu, Changzheng Wei, Jian Li, Brian Will, Ping Yu, Lu Gong, and Haibing Guan. 2019. QTLS: High-Performance TLS Asynchronous Offload Framework with Intel® QuickAssist Technology. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*.

[13] IETF RFC 7252. https://tools.ietf.org/html/rfc7252. Accessed: 2020-05-08.

[14] IETF RFC 7540. https://tools.ietf.org/html/rfc7540. Accessed: 2020-05-08.

[15] Intel QuickAssist Adaptor 8960/8970. https://www.marvell.com/products/security-solutions/nitrox-security-processors/nitrox-v.html. Accessed: 2020-05-08.

[16] Intel QuickAssist Technology. https://01.org/intel-quickassist-technology. Accessed: 2020-05-08.

[17] Takashi Isobe, Satoshi Tsutsumi, Koichiro Seto, Kenji Aoshima, and Kazutoshi Kariya. 2010. 10 Gbps Implementation of TLS/SSL Accel-erator on FPGA. In *Proceedings of the 18th International Workshop onQuality of Service (IWQoS)*.

[18] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*.

[19] S. Han S. Moon K. Jang S. Han and K. Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[20] Zia-Uddin-Ahamed Khan and Mohammed Benaissa. 2015. Throughput/Area-efficient ECC Processor Using Montgomery Point Multiplication on FPGA. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62-II, 11, 1078–1082.

[21] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, and J. Iyengar. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data (SIGCOMM)*, 183–196.

[22] Mellanox ASAP^2. https://www.mellanox.com/related-docs/products/SB_asap2.pdf. Accessed: 2020-05-08.

[23] Mellanox BlueField®-2. https://www.mellanox.com/products/bluefield2-overview. Accessed: 2020-05-08.

[24] Mellanox BlueField™ SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf. Accessed: 2020-05-08.

[25] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[26] Netronome Agilio LX SmartNICs. https://www.netronome.com/products/agilio-lx/. Accessed: 2020-05-08.

[27] nginx High performance Load Balancer, Web Server. https://www.nginx.com/. Accessed: 2020-05-08.

[28] Nitrox Security Processors. https://www.marvell.com/products/security-solutions/nitrox-security-processors.html. Accessed: 2020-05-08.

[29] Nitrox V Processsors. https://www.marvell.com/products/security-solutions/nitrox-security-processors/nitrox-v.html. Accessed: 2020-05-08.

[30] Skype. https://skype.com. Accessed: 2020-05-08.

[31] Mostafa I Soliman and Ghada Y Abozaid. 2011. FPGA Implementation and Performance Evaluation of a High Throughput Crypto Coprocessor. *Journal of Parallel and Distributed Computing (JPDC)*, 71, 8, 1075–1084.

[32] Brian Will, Andrea Grandi, and Nicolas Salhuana. 2017. Intel® QuickAssist Technology & OpenSSL-1.1.0: Performance. Technical report. Intel.

[33] Zoom. https://zoom.us. Accessed: 2020-05-08.